OMG Data Distribution Service (DDS)

DDS

DDS specification describes the **Data-Centric Publish-Subscribe (DCPS)** model for **distributed** application **communication** and integration

"Efficient and Robust Delivery of the Right Information to the Right Place at the Right Time"

What is DDS?

A **middleware protocol** and **API standard** for data-centric connectivity integrating the components of a system together

Benefits:

1. **low-latency** data connectivity
2. extreme **reliability**
3. **scalable** architecture

(**Middleware**: software layer between the operating system and applications. It enables the various components of a system to communicate and share data by letting developers focus on the specific purpose of their applications rather than the mechanics of passing information between applications and systems)

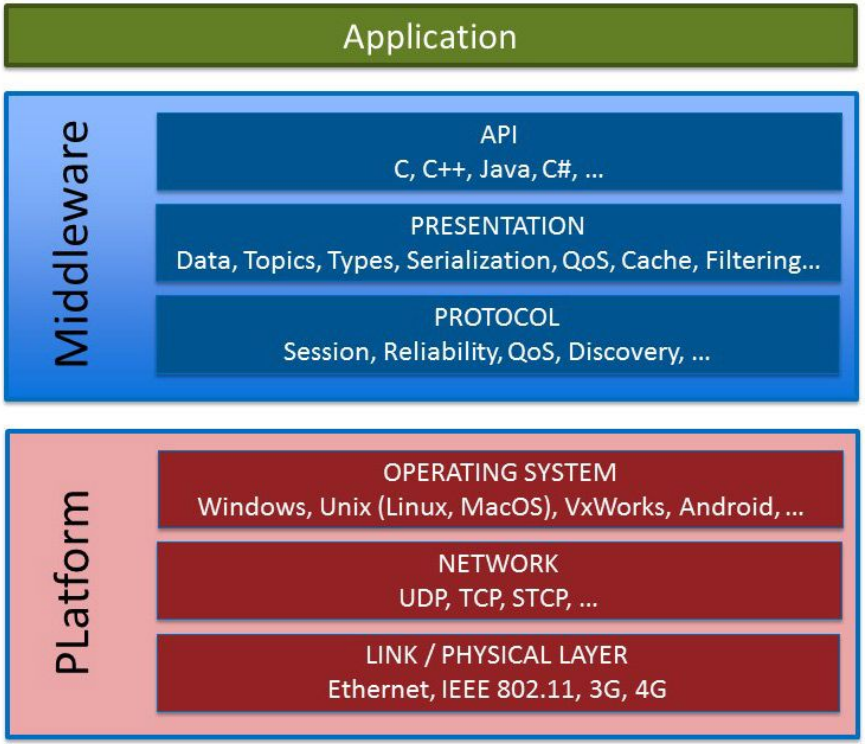
DDS Benefits

Real-time: right information is delivered at the right place at the right time all the time. Failing to deliver key information within the required deadlines can lead to life-, mission- or business-threatening situations

Dependable: thus ensuring availability, reliability, safety and integrity in spite of hardware and software failures

High-Performance: hence able to distribute very high volumes of data with very low latency

DDS Middleware



Source: https://www.dds-foundation.org/what-is-dds-3/

Why DDS

Many real-time applications need to create a data-centric communication where the published data is then available to a remote application

In this context, predictable data distribution with minimal overhead is a primary concern

→ Important to assign resources to critical requirements (QoS)

DDS Scalability

Scalable and **flexible** infrastructure supporting **thousands of publisher and subscribers**

To achieve this, **entities are decoupled** so to become easy to extend

DDS: Typed Interfaces

DDS relies on **typed interfaces** (taking into account **data type**) providing the following benefits:

1. **simple to use** → the developer directly manipulates data
2. **safe to use** → verification can be performed at compile time
3. **more efficient** → execution code can rely on specific expected data in advance (e.g., allocate resource)

Quality of Service (QoS)

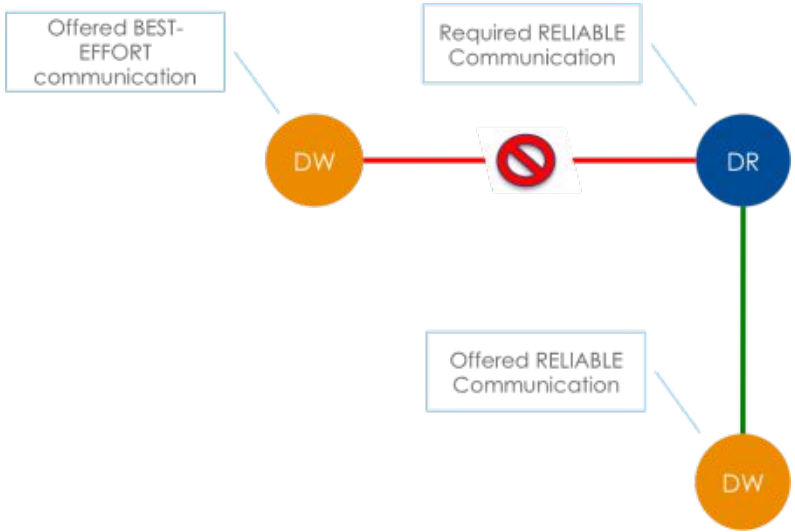
QoS is a general concept used to specify the behavior of a service

QoS settings: permits developer to indicate ‘what’ is wanted rather than ‘how’ this QoS should be achieved.

QoS policies: independent descriptions that associate a name with a value → high flexibility

Quality of Service (QoS)

Are a set of configurable parameters that control the behavior of a DDS system, such as resource consumption, fault tolerance, and communication reliability



DDS Data Memory

Distributed shared memory is difficult to implement efficiently over a network and does not offer the required scalability and flexibility

Data-Centric Publish-Subscribe (DCPS) model, has become popular in many real-time applications

A **global data space** is accessible to all interested applications becoming ***Publishers***

Applications that want to **access data space** declare their intent to become ***Subscribers***

Each time a Publisher posts new data into this global data space the middleware **propagates the information** to all interested Subscribers

Data Model

a **Data Model** defines the global data space and specifies how publishers and subscribers refer to portions of this space

Data model contains **unrelated data-structures**, each identified by a

topic and a **type**

Contains **unique identifiers** for data items

Structural information to tell **how to manipulate data**

The advantage?

The purpose of the DDS specification is to define the standardized interfaces and behaviors that enable application portability

Data-Centric Publish-Subscribe (DCPS)

DCPS allows 3 main functionalities:

1. Publishing applications can **identify data objects to publish** and provide values
2. Subscribing applications can **identify data objects to read** and access values
3. Applications can **define topics, create pub/sub entities, and attach QoS** policies

Platform Independent Model (PIM)

Platform Independent Model (PIM) is a model of a subsystem that contains **no information specific** to the computing platform or the technology that is used to realize it

Focuses on the **operation of a system while hiding the details** necessary for a particular platform. A platform independent view shows that part of the complete specification that does not change from one platform to another

PIM class

<parameter> can contain the modifier **in**, **out**, or **inout** ahead of the parameter name (default “in”)

A collection of elements of a **<type>** is identified by **<type> [ ]**

<class name>		
attributes		
<attribute name>	<attribute type>	
...	...	
operations		
<operation name>		<return type>
	<parameter>	<parameter type>
	...	...
...		...

An Example of PIM

Myclass		
attributes		
`my_attribute`	`long`	
operations		
`my_operation`		`long`
	`out: param1`	`long`
	`inout: param2`	`long`
	`param3`	`long`
	`in: param4`	`long`

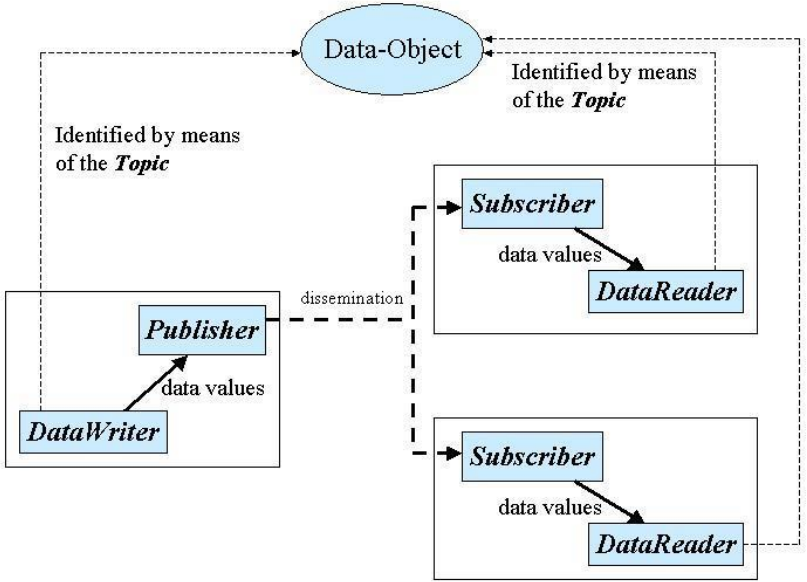
PIM Return Codes

Return codes	
OK	Successful return.
ERROR	Generic, unspecified error.
BAD_PARAMETER	Illegal parameter value.
UNSUPPORTED	Unsupported operation. Can only be returned by operations that are optional.
ALREADY_DELETED	The object target of this operation has already been deleted.
OUT_OF_RESOURCES	Service ran out of the resources needed to complete the operation.
NOT_ENABLED	Operation invoked on an ***Entity*** that is not yet enabled.
IMMUTABLE_POLICY	Application attempted to modify an immutable ***QosPolicy***.
INCONSISTENT_POLICY	Application specified a set of policies that are not consistent with each other.
PRECONDITION_NOT_MET	A pre-condition for the operation was not met.
TIMEOUT	The operation timed out.
ILLEGAL_OPERATION	An operation was invoked on an inappropriate object or at an inappropriate time (as determined by policies set by the specification or the Service implementation). There is no precondition that could be changed to make the operation succeed.
NO_DATA	Indicates a transient situation where the operation did not return any data but there is no inherent error.

DDS Information Flow

Sending: **Publisher** and **DataWriter**

Receiving: **Subscriber** and **DataReader**



Publisher

A **Publisher** is an object responsible for **data distribution** (according to **QoS**)

A **DataWriter** acts as a typed accessor to a publisher and it is the object the application must use to **communicate** to a publisher the **existence and value of data-objects** of a given type

A **publication** is defined by the association of a data-writer to a publisher. This association expresses the intent of the application to **publish the data** described by the data-writer in the context provided by the publisher

Subscriber

A **Subscriber** is an object responsible for **receiving published data** and making it available (according to **QoS**) to the receiving application.

The application must use a **typed DataReader** attached to the subscriber.

A **subscription** is defined by the **association of a data-reader with a subscriber** expressing the intent of the application to subscribe to the data described by the data-reader in the context provided by the subscriber

Topics and QoS

Topic objects conceptually fit between publications and subscriptions and represents the **unit for information that can produced or consumed** by a DDS application

A **Topic** associates a **name**, a **data-type**, and **QoS** related to the **data** itself.

Topics and **QoS** of the DataWriter/DataReader associated with Publisher/Subscriber **control the behavior** on the different sides

Topic types

Independence from a specific programming language and OS, portability and interoperability → subset of **IDL** as the formalism for describing topic types

A **topic** is made by **struct** plus a **key**

Struct can contain as many fields as you want and each field can be:

1. a **primitive** type
2. a **template** type
3. a **constructed** type

IDL Types

Primitive Types	
boolean	long
octet	unsigned long
char	long long
wchar	unsigned long long
short	float
unsigned short	double
	long double

Template Type	Example
string<length = UNBOUNDED>	string s1; string<32> s2;
wstring<length = UNBOUNDED>	wstring ws1; wstring<64> ws2;
sequence<T,length = UNBOUNDED>	sequence<octet> oseq; sequence<octet, 1024> oseq1k; sequence<MyType> mtseq; sequence<MyType, 10> mtseq10;
fixed<digits,scale>	fixed<5,2> fp; //d1d2d3.d4d5

Constructed Types	Example
enum	enum Dimension { 1D, 2D, 3D, 4D };
struct	struct Coord1D { long x;}; struct Coord2D { long x; long y; }; struct Coord3D { long x; long y; long z; }; struct Coord4D { long x; long y; long z, unsigned long long t;};

union	union Coord switch (Dimension) {
	case 1D:
	Coord1D c1d;
	case 2D:
	Coord2D c2d;
	case 3D:
	Coord3D c3d;
	case 4D:
	Coord4D c4d;
	};

Source: https://www.laas.fr/files/SLides-A_Corsaro.pdf

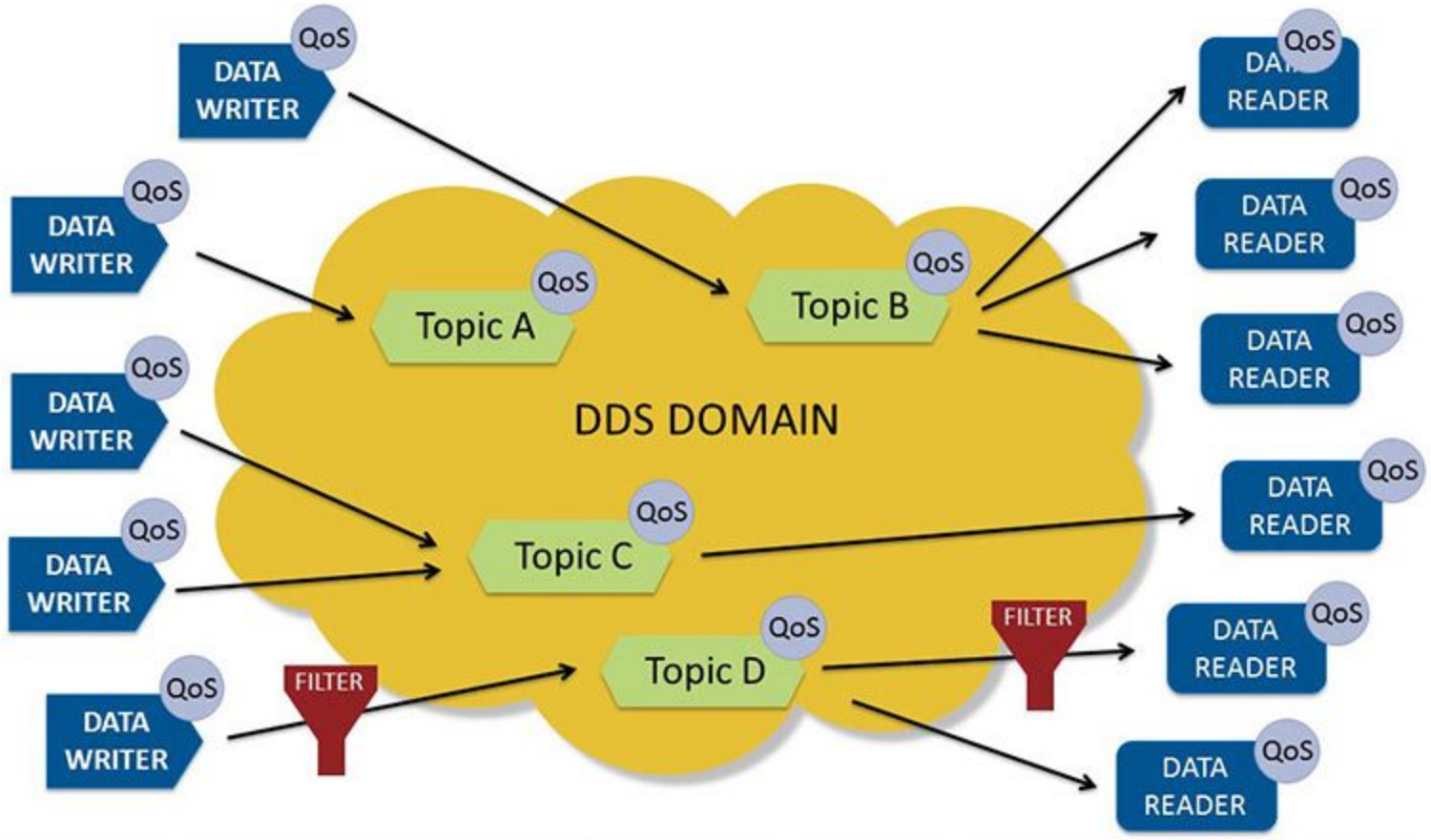
Filtering and Conditions

DDS data management also supports two ways of capturing specific data

Content-Filtered Topics: permit to subscribe to filtered topics and receiving only values matching it

Query Conditions: simply queries data received and available on the existing reader cache

Operator	Description
=	Equal
<>	Not Equal
>	Greater Than
<	Less Than
>=	Greater then equal
<=	Less than equal
BETWEEN	Between and inclusive range
LIKE	Search for a pattern



Source: https://www.dds-foundation.org/what-is-dds-3/

Domain

A Domain is a communication context, which provides a virtual environment, encapsulating different concerns and thereby optimizing communications.

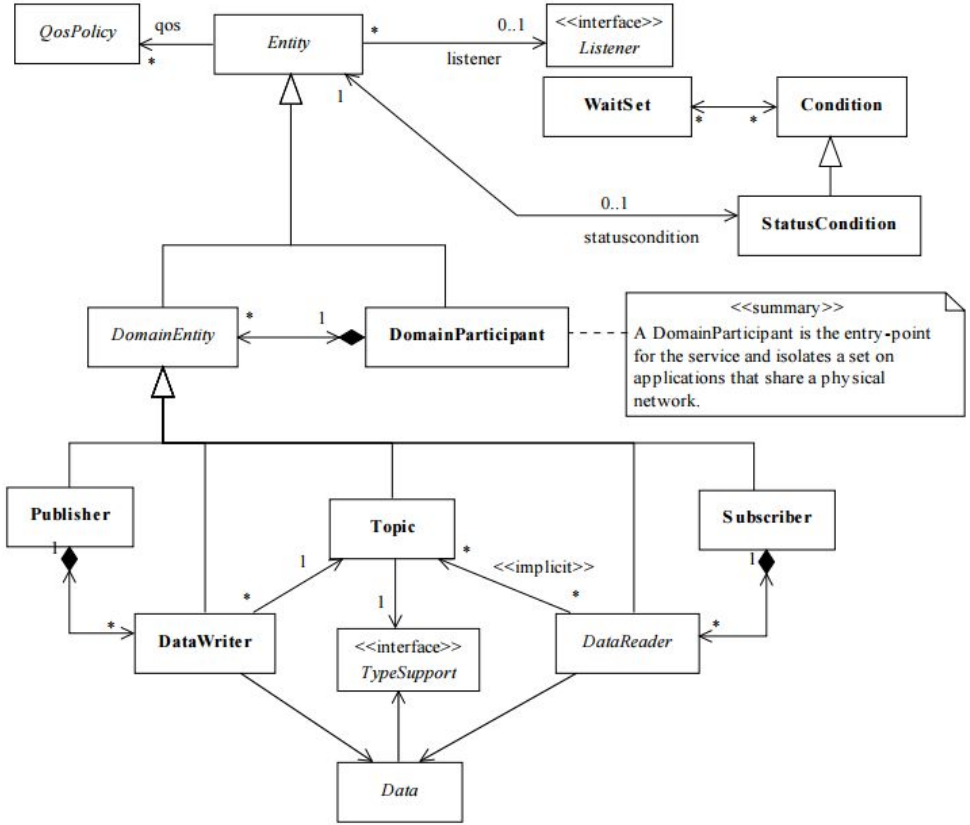
DDS applications send and receive data within a domain which also isolates participants associated with different domains

Scoping Information

DDS provides two mechanism for scoping information:

1. **Domains:** A domain establishes a **virtual network linking all the DDS applications** that have joined it. No communication can ever happen across domains unless explicitly mediated by the user application
2. **Partitions:** Domains can be further organized into partitions, where each partition represent a logical grouping of topics

DDS Conceptual Model



DDS Conceptual Model

Communication objects patterns:

1. each entity supports specialized **QoS** policies
2. each entity accepts a specialized **Listener** as mechanism to notify the application of events such as the arrival of data corresponding to a subscription, violation of QoS
3. each entity accepts a **StatusCondition** providing an alternative communication style between the middleware and the application

All the entities are attached to a **DomainParticipant** which represents the local membership of an application in a domain

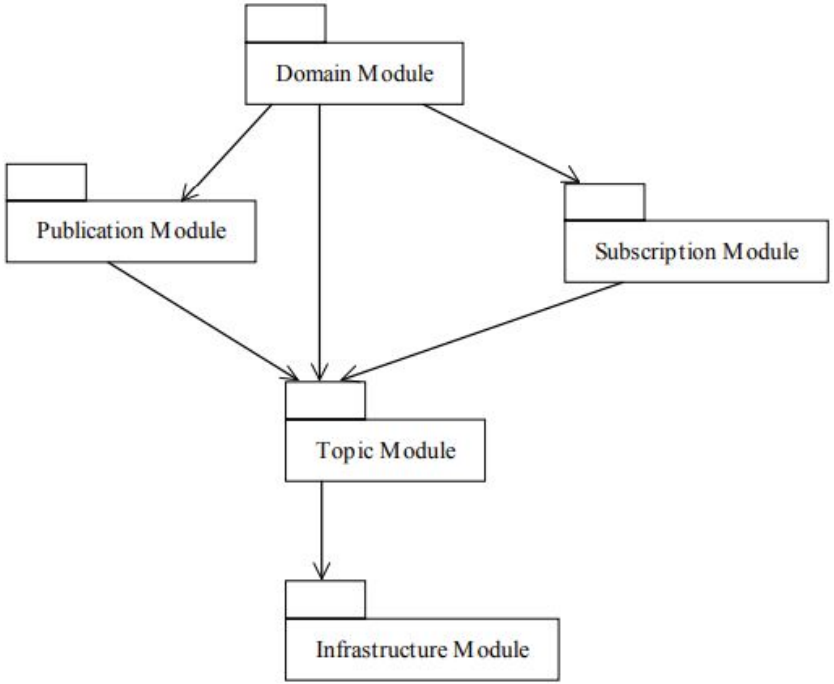
A **Domain links all the applications** and represents a communication plane, only pubs/subs attached to the same domain can interact

Status condition

Status conditions are mechanisms that allow applications to monitor the state of DDS entities (such as Data Writers, Data Readers, Publishers, and Subscribers) by setting specific triggers or callbacks for key events.

Status conditions help in managing and responding to changes or updates in the system, such as new data availability, connection status, or errors.

DDS PIM


Domain Module
Publication Module
Subscription Module
Topic Module
Infrastructure Module

DCPS Modules

The DCPS is comprised of five modules:

1. **Infrastructure Module:** defines the abstract classes and the interfaces that are refined by the other modules. It also provides support for the interaction with the middleware

2. **Domain Module:** contains the **DomainParticipant** class that acts as an entry-point of the Service and acts as a factory for many of the classes. The DomainParticipant also acts as a container for the other objects that make up the Service

3. **Topic-Definition Module:** contains the **Topic**, **ContentFilteredTopic**, and **MultiTopic** classes, the **TopicListener** interface, and more generally, all that is needed by the application to define Topic objects and attach QoS policies to them

4. **Publication Module:** contains the Publisher and DataWriter classes as well as the PublisherListener and DataWriterListener interfaces, and more generally, all that is needed on the publication side

5. **Subscription Module:** contains the **Subscriber**, **DataReader**, **ReadCondition**, and QueryCondition classes, as well as the **SubscriberListener** and **DataReaderListener** interfaces, and more generally, all that is needed on the subscription side

Entity Class

Entity is the abstract base class for all the DCPS objects that support QoS policies, a listener and a status condition

1. set_qos
2. get_qos
3. set_listener
4. get_listener
5. get_statuscondition
6. get_status_change
7. enable
8. get_instance_handle

Entity		
no attributes		
operations		
abstract set_qos		ReturnCode_t
	qos_list	QosPolicy []
abstract get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
abstract set_listener		ReturnCode_t
	a_listener	Listener
	mask	StatusKind []
abstract get_listener		Listener
get_statuscondition		StatusCondition
get_status_changes		StatusKind []
enable		ReturnCode_t
get_instance_handle		InstanceHandle_t

WaitSet Class

A **WaitSet** object allows an application to wait until one or more of the attached **Condition** objects has a trigger_value of **TRUE** or else until the timeout expires

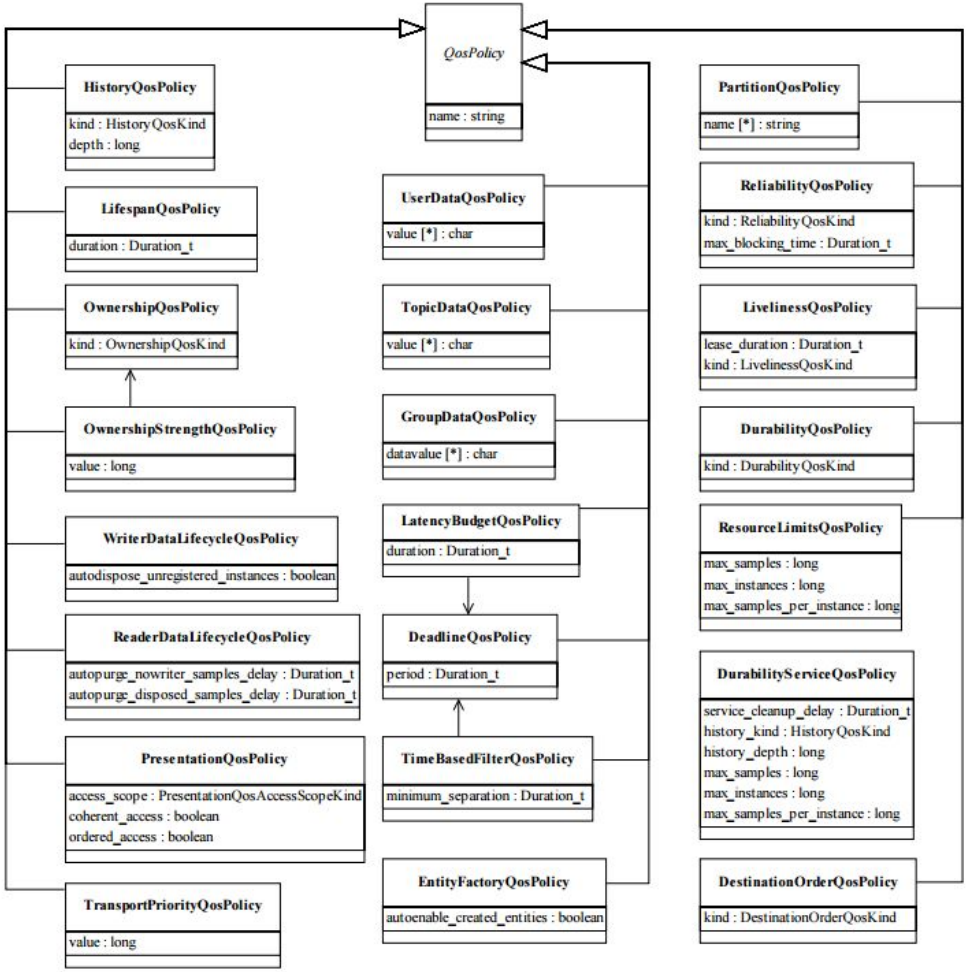
1. attach_condition
2. detach_condition
3. wait
4. get_conditions

WaitSet		
no attributes		
operations		
attach_condition		ReturnCode_t
	a_condition	Condition
detach_condition		ReturnCode_t
	a_condition	Condition
wait		ReturnCode_t
	inout: active_conditions	Condition []
	timeout	Duration_t
get_conditions		ReturnCode_t
	inout: attached_conditions	Condition []

Supported QoS

DDS relies on Quality of Service which is a set of characteristics that controls some aspect of the behavior of the DDS Service.

QoS is comprised of individual QoS policies


QosPolicy
name : string
HistoryQosPolicy
kind : HistoryQosKind
depth : long
LifespanQosPolicy
duration : Duration_t
OwnershipQosPolicy
kind : OwnershipQosKind
OwnershipStrengthQosPolicy
value : long
WriterDataLifecycleQosPolicy
autodispose_unregistered_instances : boolean
ReaderDataLifecycleQosPolicy
autopurge_nowriter_samples_delay : Duration_t
autopurge_disposed_samples_delay : Duration_t
PresentationQosPolicy
access_scope : PresentationQosAccessScopeKind
coherent_access : boolean
ordered_access : boolean
TransportPriorityQosPolicy
value : long
UserDataQosPolicy
value [*] : char
TopicDataQosPolicy
value [*] : char
GroupDataQosPolicy
datavalue [*] : char
LatencyBudgetQosPolicy
duration : Duration_t
DeadlineQosPolicy
period : Duration_t
TimeBasedFilterQosPolicy
minimum_separation : Duration_t
EntityFactoryQosPolicy
autoenable_created_entities : boolean
PartitionQosPolicy
name [*] : string
ReliabilityQosPolicy
kind : ReliabilityQosKind
max_blocking_time : Duration_t
LivelinessQosPolicy
lease_duration : Duration_t
kind : LivelinessQosKind
DurabilityQosPolicy
kind : DurabilityQosKind
ResourceLimitsQosPolicy
max_samples : long
max_instances : long
max_samples_per_instance : long
DurabilityServiceQosPolicy
service_cleanup_delay : Duration_t
history_kind : HistoryQosKind
history_depth : long
max_samples : long
max_instances : long
max_samples_per_instance : long
DestinationOrderQosPolicy
kind : DestinationOrderQosKind

Some Significant QoS

Reliability indicates the level of guarantee provided to the subscribers in the data exchange phase

1. **Reliable**: middleware guarantees that all the samples in the DataWriter will be delivered to all the DataReader
2. **Best Effort**: it is acceptable to not retry the samples propagation

QosPolicy	Value	Meaning	Concerns	RxO	Changeable
RELIABILITY	A "kind": RELIABLE, BEST_EFFORT and a duration "max_blocking_time"	Indicates the level of reliability offered/ requested by the Service.	Topic, DataReader, DataWriter	Yes	No
	RELIABLE	Specifies the Service will attempt to deliver all samples in its history. Missed samples may be retried. In steady-state (no modifications communicated via the ***DataWriter***) the middleware guarantees that all samples in the ***DataWriter*** history will eventually be delivered to all the ***DataReader***[1] objects. Outside steady state the HISTORY and RESOURCE_LIMITS policies will determine how samples become part of the history and whether samples can be discarded from it. This is the default value for ***DataWriters***.			
	BEST_EFFORT	Indicates that it is acceptable to not retry propagation of any samples. Presumably new values for the samples are generated often enough that it is not necessary to re-send or acknowledge any samples. This is the default value for ***DataReaders*** and ***Topics***.			
	max_blocking_time	The value of the *max_blocking_time* indicates the maximum time the operation *DataWriter::write* is allowed to block if the *DataWriter* does not have space to store the value written. The default max_blocking_time=100ms.			

Some Significant QoS

History controls which value should be delivered in terms of time

1. **Keep Last**: DDS keeps only the most recent sample
2. **Keep All**: DDS keeps all the samples

HISTORY	A "kind": KEEP_LAST, KEEP_ALL And an optional integer "depth"	Specifies the behavior of the Service in the case where the value of a sample changes (one or more times) before it can be successfully communicated to one or more existing subscribers. This QoS policy controls whether the Service should deliver only the most recent value, attempt to deliver all intermediate values, or do something in between. On the publishing side this policy controls the samples that should be maintained by the ***DataWriter*** on behalf of existing ***DataReader*** entities. The behavior with regards to a ***DataReader*** entities discovered after a sample is written is controlled by the DURABILITY QoS policy. On the subscribing side it controls the samples that should be maintained until the application "takes" them from the Service.	Topic, DataReader, DataWriter	No	No
	KEEP_LAST and optional integer "depth"	On the publishing side, the Service will only attempt to keep the most recent "depth" samples of each instance of data (identified by its key) managed by the ***DataWriter***. On the subscribing side, the ***DataReader*** will only attempt to keep the most recent "depth" samples received for each instance (identified by its ***key***) until the application "takes" them via the ***DataReader's take*** operation. KEEP_LAST is the default ***kind***. The default value of ***depth*** is 1. If a value other than 1 is specified, it should be consistent with the settings of the RESOURCE_LIMITS QoS policy.			
	KEEP_ALL	On the publishing side, the Service will attempt to keep all samples (representing each value written) of each instance of data (identified by its ***key***) managed by the ***DataWriter*** until they can be delivered to all subscribers. On the subscribing side, the Service will attempt to keep all samples of each instance of data (identified by its ***key***) managed by the ***DataReader***. These samples are kept until the application "takes" them from the Service via the ***take*** operation. The setting of ***depth*** has no effect. Its implied value is LENGTH_UNLIMITED[2].			

Some Significant QoS

Durability controls data variability

1. **Volatile**: no need to keep data for late joinings
2. **Transient Local**: data availability for late joinings is tied to data writer availability
3. **Transient**: data availability outlives the data writer
4. **Persistent**: data availability otulives system restart

DURABILITY	A "kind": VOLATILE, TRANSIENT_ LOCAL, TRANSIENT, or PERSISTENT	This policy expresses if the data should 'outlive' their writing time.	Topic, DataReader, DataWriter	Yes	No
	VOLATILE	The Service does not need to keep any samples of data-instances on behalf of any *DataReader* that is not known by the *DataWriter* at the time the instance is written. In other words the Service will only attempt to provide the data to existing subscribers. This is the default kind.			
	TRANSIENT_ LOCAL, TRANSIENT	The Service will attempt to keep some samples so that they can be delivered to any potential late-joining *DataReader*. Which particular samples are kept depends on other QoS such as HISTORY and RESOURCE_LIMITS. For TRANSIENT_LOCAL, the service is only required to keep the data in the memory of the *DataWriter* that wrote the data and the data is not required to survive the *DataWriter*. For TRANSIENT, the service is only required to keep the data in memory and not in permanent storage; but the data is not tied to the lifecycle of the *DataWriter* and will, in general, survive it. Support for TRANSIENT kind is optional.			

QoS Inconsistency

QoS Policies may not be consistent with others!

Policies are added to verified ones to catch inconsistent states, how to manage this?

Create Policies compatible on Publisher and Subscriber side following the **subscriber-requested, publisher-offered pattern**

Subscriber-requested, publisher-offered pattern

Subscriber specifies a requested value for a QoS Policy

Publisher specifies an offered value for that QoS Policy

If the two policies are compatible then the connection is established

RxO property indicates compatibility need:

- **Yes**: policy can be set at both sides and they are dependent
- **No**: policy can be set at both sides but they are independent
- **N/A**: policy can be set only at one side

Listeners, Conditions, and Wait-sets

Listeners and **conditions** (in conjunction with wait-sets) are two alternative mechanisms that allow the application to **be aware of changes** in the DCPS communication status

Communication Status

The communication statuses whose changes can be communicated to the application depend on the **Entity**

Entity	Status Name	Meaning
Topic	INCONSISTENT_TOPIC	Another topic exists with the same name but different characteristics.
Subscriber	DATA_ON_READERS	New information is available.
DataReader	SAMPLE_REJECTED	A (received) sample has been rejected.
	LIVELINESS_CHANGED	The liveliness of one or more ***DataWriter*** that were writing instances read through the ***DataReader*** has changed. Some ***DataWriter*** have become "active" or "inactive."
	REQUESTED_DEADLINE_MISSED	The deadline that the ***DataReader*** was expecting through its ***QosPolicy*** DEADLINE was not respected for a specific instance.
	REQUESTED_INCOMPATIBLE_QOS	A ***QosPolicy*** value was incompatible with what is offered.
	DATA_AVAILABLE	New information is available.
	SAMPLE_LOST	A sample has been lost (never received).
	SUBSCRIPTION_MATCHED	The *DataReader* has found a *DataWriter* that matches the *Topic* and has compatible QoS, or has ceased to be matched with a ***DataWriter*** that was previously considered to be matched.
DataWriter	LIVELINESS_LOST	The liveliness that the ***DataWriter*** has committed through its ***QosPolicy*** LIVELINESS was not respected; thus ***DataReader*** entities will consider the ***DataWriter*** as no longer "active."
	OFFERED_DEADLINE_MISSED	The deadline that the ***DataWriter*** has committed through its ***QosPolicy*** DEADLINE was not respected for a specific instance.
	OFFERED_INCOMPATIBLE_QOS	A ***QosPolicy*** value was incompatible with what was requested.
	PUBLICATION_MATCHED	The *DataWriter* has found *DataReader* that matches the *Topic* and has compatible QoS, or has ceased to be matched with a *DataReader* that was previously considered to be matched.

Catch Status Change

Associated with each one of an Entity's communication status is a logical flag. This flag indicates whether that particular communication status has changed since the last time the status was 'read' by the application

Access through Listeners

Listeners provide a mechanism for the middleware to **asynchronously alert** the application of the occurrence of relevant **status changes**

All Entity support a listener, which type of which is **specialized** to the specific type of the related Entity

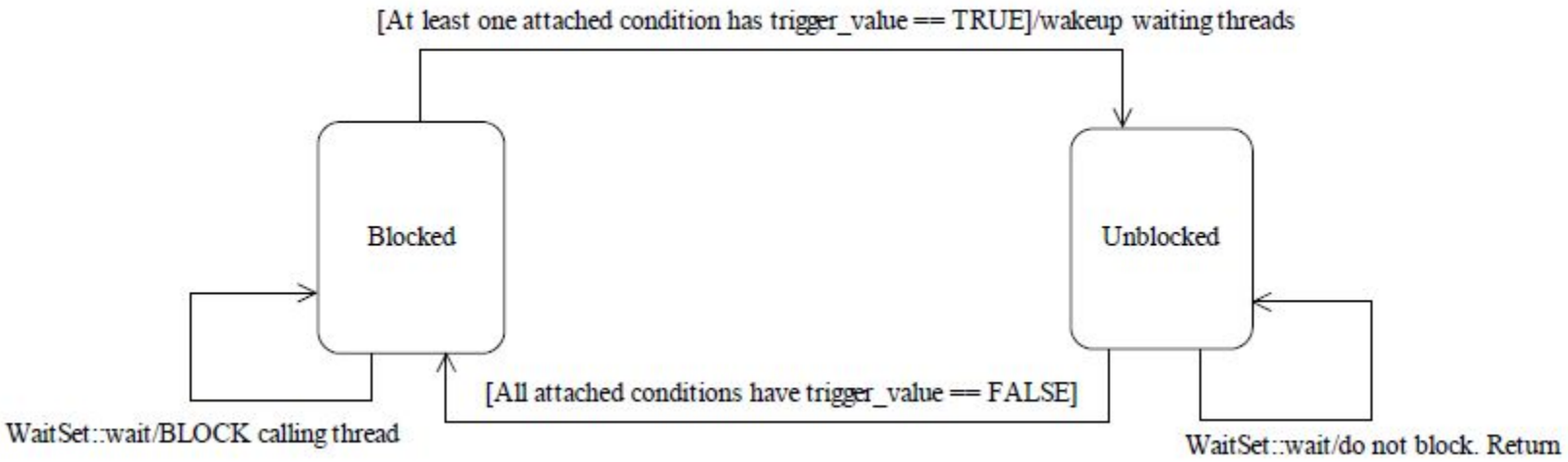
Each dedicated **listener presents a list of operations** that correspond to the relevant communication status changes

Conditions and Wait-sets

conditions (in conjunction with wait-sets) provide an alternative mechanism to allow the middleware to communicate communication status changes to the application

1. The application indicates which relevant information it wants to get, by creating **Condition objects** and attaching them to a **WaitSet**
2. It then waits on that WaitSet until the **trigger_value** of one or several Condition objects become **TRUE**
3. It then uses the result of the wait to actually get the information by calling

Conditions and Wait-sets



Built-in Topics

The middleware must discover and possibly keep track of the presence of remote entities such as a new participant in the domain

→ **Built-in topics** and corresponding **DataReader objects** that can then be used by the application

→ No need to introduce a new API to access information

A Real Example

Who: V-Charge Research program with industrial (Bosch and Volkswagen AG) and academic (ETH Zurich and the universities of Braunschweig, Oxford and Parma) partners

Goal: automatically park an electric car in a large parking garage with centimeter accuracy at an induction-based charging station.

Setup: Automated and driverless parking in a multi-story car park cannot rely on GPS to facilitate navigational guidance. Instead, a suite of 2 stereo cameras, 4 mono-cameras and 12 or more sensors in the car compare their information against a pre-mapped car park layout to determine location. For safety, the system must recognize other static and moving objects, and adjust automated drive control appropriately

Source: https://info.rti.com/hubfs/Collateral_2017/Customer_Snapshots/RTI_Customer-Snapshot_60016_Volkswagen_V6_0718.pdf

A Real Example

Challenge: researchers can stay focused on application level issues rather than dealing with system architecture problems created by changes in systems integration. To enable the compute intensive sensor integration across the distributed environment, the team needed a common way to deliver data between modules and sub-systems that integrated with existing software toolchains.

Solution: RTI Connext DDS as the integration middleware. DDS enables the decoupling of the applications from the underlying communication infrastructure. This means that any process or module can be moved around the network completely unchanged. DDS discovers the modules new location at boot time and routes communication as needed. No IP reconfiguration or network interface changes are required by the application. DDS includes a Quality of Service (QoS) capability that ensures that the data needs of each and every communicating process and module are being met. If they are not, the discrepancy between publisher and subscriber is noted and both applications are informed. No application module needs to know about any other – it just needs to know it requires specific data within certain timing and delivery constraints and to know when those constraints are not being met.